华中科技大学

计算机视觉课程实验报告

实验三：基于卷积神经网络的两位数字比较

姓　　名：　　李嘉鹏

学　　院：　　计算机科学与技术学院

专　　业：　　数据科学与大数据技术

班　　级：　　大数据 2101 班

学　　号：　　U202115652

指导教师：　　刘康

2023 年 12 月 25 日

目　录

实验三 基于卷积神经网络的两位数字比较

1.1 Introduction（实验简介）

1.1.1 Background（实验背景）

卷积神经网络也是计算机视觉领域中常见的一种网络结构，它属于前馈神经网络的一种。卷积神经网络有三个结构上的特性：局部连接、权重共享、空间或时间上的次采样。卷积神经网络的基本结构如图 1 所示。

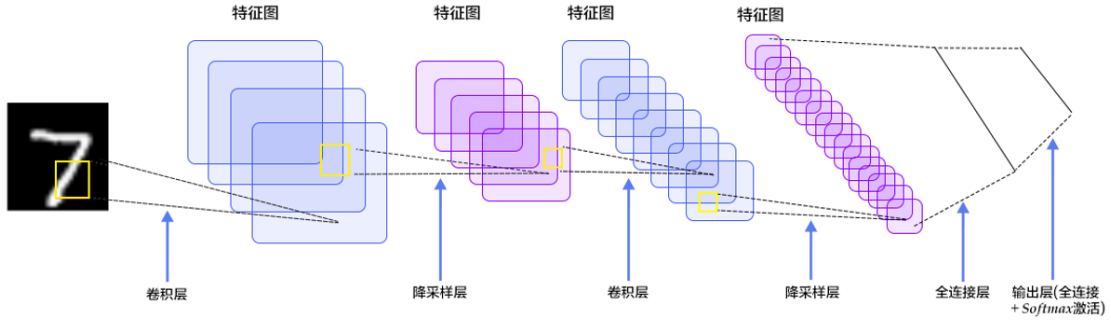

图 1：卷积神经网络结构

整个卷积神经网络主要分为卷积层（进行图像的特征提取）、池化层（减少计算量并实现平移、旋转、尺度不变性）、全连接层（将网络提取到的特征进行整合）。经典的卷积神经网络包括 ResNet、LeNet、AlexNet 和 VGG 等。

本实验中，我沿用了实验二中的 ResNet 模块，通过修改输入维度和输出维度，实现了目标功能。

1.1.2 ResNet 介绍

ResNet 是一种较新的神经网络结构，它引入了残差单元来解决退化问题。其瓶颈残差模块一般依次由 1×1、3×3、1×1 这三个卷积层堆积而成，其中 1×1 卷积能起到降维或升维作用，从而使 3×3 的卷积在相对较低维度的输入上进行，提高计算效率。残差模块的构成如图 2 所示。

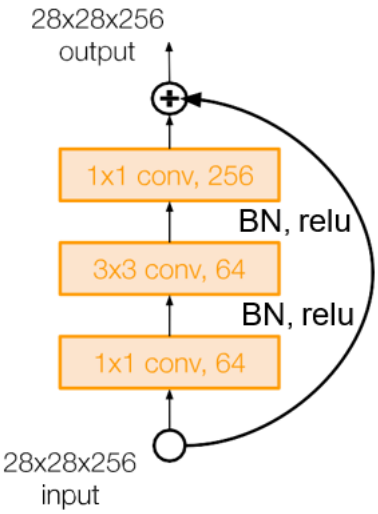

图 2：ResNet 残差模块构成示意图

该残差模块有助于解决梯度消失和退化问题，提高模型训练的效果。

1.1.3 Motivation（实验目的）

本实验要求设计一个卷积神经网络，输入为两张 MNIST 手写体数字图片，如果两张图片为同一个数字（非同一张图片），输出为 1，否则为 0。需要从 MNIST 数据集的训练集中选取 10%作为本实验的训练图片，从 MNIST 数据集的测试集中选取 10%作为本实验的测试图片，并将这两部分图片经过适当处理后形成一定数量的用于本实验的训练集和测试集。

最终，还需要给出每一轮 mini-batch 训练后的模型在训练集和测试集上的损失，以及模型的准确率。

1.2 Proposed Method（实验设计与步骤）

1.2.1 Network Architecture of ResNet（ResNet 模块架构）

我采用的深度学习框架为 Tensorflow。

这里仍然使用了实验二中的 ResNet 模块（ResNetBlock 类），在构造函数中定义了三个卷积层和三个批量正则化层，分别为 conv1、bn1、conv2、bn2、conv3 和 bn3。在 call 方法中，定义了层的前向传播，通过依次调用这些层，将输入数据进行卷积、批量正则化和残差连接的操作，最后返回输出，代码如下所示。

```
class ResNetBlock(layers.Layer):
    def __init__(self, filters):
        super(ResNetBlock, self).__init__()
        self.conv1 = layers.Conv2D(filters, (1, 1), activation='relu')
        self.bn1 = layers.BatchNormalization()
        self.conv2 = layers.Conv2D(filters, (3, 3), activation='relu',
padding='same')
        self.bn2 = layers.BatchNormalization()
        self.conv3 = layers.Conv2D(filters, (1, 1))
        self.bn3 = layers.BatchNormalization()

    def call(self, inputs):
        x = self.conv1(inputs)
        x = self.bn1(x)
        x = self.conv2(x)
        x = self.bn2(x)
        x = self.conv3(x)
        x = self.bn3(x)
        x = layers.Add()([inputs, x])
        x = layers.Activation('relu')(x)
        return x
```

ResNet 模块内部的主要架构是：

（1）Conv2D 卷积层

卷积层 conv1、conv2 和 conv3 通过卷积操作提取输入数据的特征。其中，conv1 和 conv3 使用 1x1 的卷积核进行卷积操作，而 conv2 使用 3x3 的卷积核进行卷积操作，并且通过在外围 padding 的方式确保输出特征图和输入特征图的大小相同。三个卷积层的步长均为 1。这三层共同构成 ResNet 的残差模块。

（2）BatchNormalization 批量正则化层

批量正则化层 bn1、bn2 和 bn3 对卷积层的输出进行统一的批量正则化操作，加速训练过程，提高在测试集上的泛化能力。其本质就是在每一层输入时插入了一个归一化处理（归一化后数据的均值 $\overline{\mu}=0$，方差 $\sigma^2=1$），然后再进入网络的下一层。

（3）Add 层和 Activation 层

在 ResNet 模块的最后，将输入数据与经过卷积和批量正则化处理后的输出进行残差连接（add），将之前的输入直接与当前层的输出相加，从而跳过一部分网络层，把前面的特征传递给后面的层。最后，通过使用 relu 激活函数来确保输出是非线性的。需要注意的是，这里把激活函数放在最后一步，是因为如果最后一次激活函数放在第三个卷积层之后，那么残差连接之后的特征将直接受到激活函数的影响，可能导致残差信息被改变或丢失。

1.2.2 Network Architecture of CNN（卷积神经网络架构）

定义好 ResNet 模块后，现在可以直接使用封装好的 ResNet 模块实现一个完整的神经网络架构，在代码实现中我沿用了实验二中的类 CNNModel。

在 CNNModel 的构造函数中，我首先定义了卷积层 conv1、批量正则化层 bn1、最大池化层 maxpool 以及两个 ResNet 模块 resnet_block1 和 resnet_block2。在 call 方法中，我按顺序调用这些层，对输入数据进行卷积、批量正则化和残差连接操作，最后通过扁平化层 flatten 和全连接层 fc1 得到特征向量，再通过 Dropout 层 dropout 进行随机失活，最终经过输出层 fc2 得到结果，即预测出的概率分布。具体代码如下所示。

```
class CNNModel(tf.keras.Model):
    def __init__(self):
        super(CNNModel, self).__init__()
        self.conv1 = layers.Conv2D(32, (3, 3), activation='relu',
input_shape=(28, 28, 2))  # 卷积层
        self.bn1 = layers.BatchNormalization()  # 批量正则层
        self.maxpool = layers.MaxPooling2D((2, 2))  # 最大池化层
        self.resnet_block1 = ResNetBlock(32)  # ResNet 模块 1
        self.resnet_block2 = ResNetBlock(32)  # ResNet 模块 2
        self.flatten = layers.Flatten()  # 扁平化层
        self.fc1 = layers.Dense(64, activation='relu')  # 全连接层
        self.dropout = layers.Dropout(0.5) # Dropout 层
        self.fc2 = layers.Dense(2, activation='relu') # 全连接层（也就是输
出层）
```

```
    def call(self, inputs):
        x = self.conv1(inputs)
        x = self.bn1(x)
        x = self.maxpool(x)
        x = self.resnet_block1(x)
        x = self.resnet_block2(x)
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.dropout(x)
        output = self.fc2(x)
        return output
```

神经网络架构中各部分的特点和作用分别是：

（1）输入层

输入的图像大小为 28x28 像素，通道数为 2（因为 MNIST 数据集为单色图像，输入为两张图片，每张图片对应一个灰度值，二者叠加后，新的输入为 2 通道）。

（2）卷积层

包含 32 个大小为 3x3 的卷积核，负责提取图像的特征。

（3）批量正则化层

对卷积层的输出进行批量正则化，加速收敛过程。

（4）最大池化层

进行 2x2 的最大池化操作，对特征图进行下采样，在减少数据量的同时保留关键信息。

（5）ResNet 模块

每个 ResNet 模块（ResNetBlock）由三个卷积层和一个残差连接组成，具体架构见上面的定义。特征图经过 ResNet 后输出的图像大小仍为 28x28。

（6）扁平化层

将前面输出的特征图展平为一维向量，为全连接层做准备。

（7）全连接层

包含 64 个神经元，主要负责将网络提取到的特征整合到一起。

（8）Dropout 层

Dropout 正则化，随机丢弃一部分神经元，用于避免过拟合的问题。其基本步骤如图 3 所示，主要是不断重复以下两步：①随机删掉网络中一定比例的隐藏神经元，输入输出神经元保持不变；②把输入 x 通过修改后的网络前向传播，然后对修改后的神经元进行参数更新。

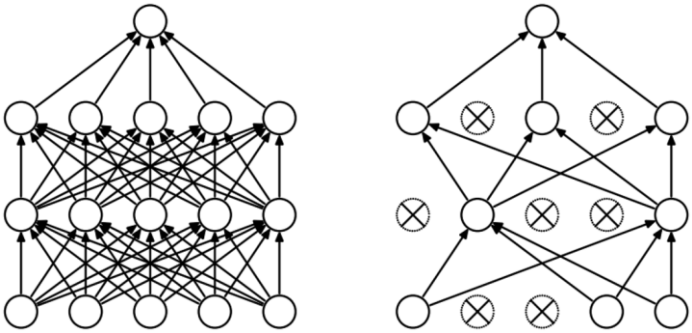

图 3：Dropout 正则化步骤示意图

（9）输出层（同时也是全连接层）

输出层只有 2 个神经元。这是一个二分类问题，也就是输入的两张图片的数字相同时输出 1，数字不同时输出 0，因此采用 relu 激活函数用于输出预测结果的概率分布。最终，概率最大的神经元对应的数字将被作为预测结果输出。

综合上面的分析，一种可行的神经网络架构如图 4 所示，各层的参数在左侧列出。网络架构图使用 NN-SVG[4]绘制。

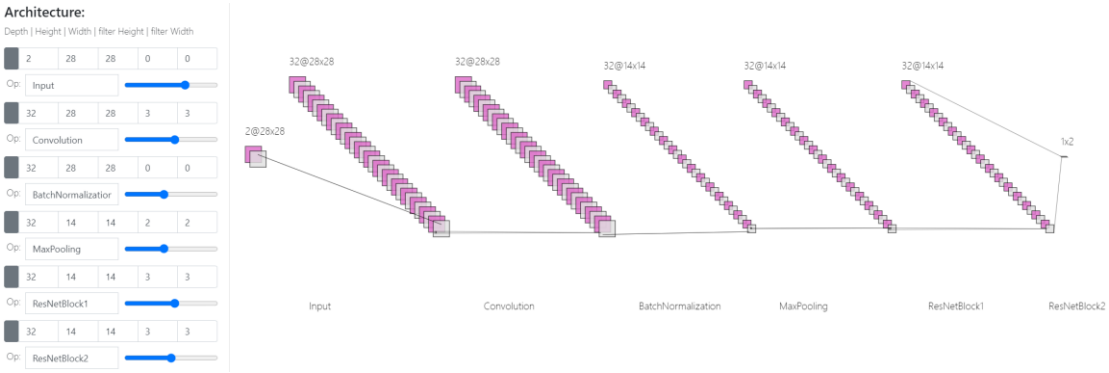

图 4：一种使用 ResNet 模块的卷积神经网络架构示意图（实验三）

由于分类任务中最常用的损失函数为交叉熵损失函数，因此本实验中我使用了 sparse_categorical_crossentropy 作为损失函数来适应二分类问题。同时，我还使用了 adam 作为优化器。训练模型时每个 epoch 包含多个 mini-batch，每个 mini-batch 中包含的训练样本数量为 batch_size。最终使用新产生的测试集来验证模型。代码如下所示。

```
model.compile(optimizer='adam',
```

```
            loss='sparse_categorical_crossentropy',
            metrics=['accuracy'])

history = model.fit(new_x_train, new_y_train, epochs=8, batch_size=8000,
class_weight=class_weight, callbacks=[LossHistory((new_x_train,
new_y_train), (new_x_test, new_y_test))])
```

1.2.3 训练集与测试集构成

对于数据集的划分，我首先下载了 MNIST 数据集，然后分别从其训练集和测试集中选取 10%作为本实验的训练图片和测试图片，数量分别为 train_samples 和 test_samples。

```
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

train_samples = int(len(x_train) * 0.1)
test_samples = int(len(x_test) * 0.1)
```

为了确保数据集的随机分布，我使用 np.random.choice 在下载下来的原始训练集和测试集中随机选取指定数量的图片作为本实验中所用的图片。

```
train_indices = np.random.choice(len(x_train), train_samples,
replace=False)
test_indices = np.random.choice(len(x_test), test_samples,
replace=False)

x_train = x_train[train_indices] / 255.0
y_train = y_train[train_indices]
x_test = x_test[test_indices] / 255.0
y_test = y_test[test_indices]
```

接下来，分别根据选取出来的训练图片和测试图片产生全新的训练集和测试集。以训练集为例，这里我使用两两遍历的方式，逐一比对每一对训练图片的数字是否相同，并按照 0-1 交替的顺序将其加入训练集中。例如：

- **第一次**：比对图片 1、图片 2，若二者代表的数字不同（标签为 0），则将其加入训练集；
- **第二次**：比对图片 1、图片 3，若二者代表的数字相同（标签为 1），则将其加入训练集；
- **第三次**：比对图片 1、图片 4，若二者代表的数字不同（标签为 0），则将其加入训练集，等等。

如果不满足当前寻找的标签条件，则跳过这组图片对，继续比对下一组图片

对是否满足条件。

当标签为 0 和 1 的数据数量均达到 50000 条时，训练集构造完成。此时训练集的标签分布为[0, 1, 0, 1, 0, 1, …, 0, 1]，**分布规律性太强，不满足随机均匀分布的要求，因此最后还需要将其进行随机打乱**。

```
# 处理训练集
label0 = 0
label1 = 0
flag = 0 # 当前寻找的标签
new_x_train = []
new_y_train = []
for i in range(0, len(x_train)-1):
    for j in range(i, len(x_train)-1):
        if flag == 1 and y_train[i] == y_train[j] and label1 < 50000:
            new_x_train.append([x_train[i], x_train[j]])
            new_y_train.append(1)
            label1 += 1
            flag = 0
        if flag == 0 and y_train[i] != y_train[j] and label0 < 50000:
            new_x_train.append([x_train[i], x_train[j]])
            new_y_train.append(0)
            label0 += 1
            flag = 1
        if label1 == 50000 and label0 == 50000:
            break

new_x_train = np.array(new_x_train)
new_y_train = np.array(new_y_train)

np.random.seed(0)
random_index = np.random.permutation(len(new_x_train))
new_x_train = new_x_train[random_index]
new_y_train = new_y_train[random_index]
```

同理，产生新的测试集的方法与上面一致。最终，使用 count_0 和 count_1 输出训练集中标签为 0 和 1 的数据条数，并在数据集上使用 shape 函数输出训练集和测试集的形状。结果如图 5 所示。

```
count_0 = np.count_nonzero(new_y_train == 0)
count_1 = np.count_nonzero(new_y_train == 1)

print("新的训练集中标签为 0 的数量：", count_0)
print("新的训练集中标签为 1 的数量：", count_1)
print("新的训练集数据形状：", new_x_train.shape)
```

```
print("新的训练集标签形状：", new_y_train.shape)
print("新的测试集数据形状：", new_x_test.shape)
print("新的测试集标签形状：", new_y_test.shape)
```

```
新的训练集中标签为0的数量： 50000
新的训练集中标签为1的数量： 50000
新的训练集数据形状： (100000, 2, 28, 28)
新的训练集标签形状： (100000,)
新的测试集数据形状： (4000, 2, 28, 28)
新的测试集标签形状： (4000,)
```

图 5：训练集与测试集构成相关信息

从上图可以发现，**训练集和测试集的维度都是 28×28×2，训练集有 100000 条数据，测试集有 4000 条数据**，符合实验要求；并且**训练集中 0 和 1 的数据条数一样（数据平衡）**，有助于提高模型的泛化性。如果要观察某一条训练集或测试集的数据，可以将其打印出来，显然输入为两张图片，如图 6 所示。

```
new_x_train[0]
[[[0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  ...
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]]

 [[0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  ...
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]]]
```

图 6：训练集数据 new_x_train[0]

整个打印结果是一个三维数组，表示图像的像素矩阵。第一个维度对应图像的行数（等于 28），第二个维度对应图像的列数（等于 28），第三个维度对应图像的通道数（等于 2）。每个元素均为归一化后的浮点数，用于表示对应位置的灰度值，范围在 0~1 之间，数值越大表示亮度越高。

1.2.4 模型的评估

最后，在检验模型效果时，为了输出模型在二分类中每一类的准确率，我首先获取了模型对所有 case 的预测概率，并找出概率最大的结果（0 或 1）作为预测结果。然后，我使用了 sklearn 库中的 classification_report 函数计算每个类别的评估指标，可以输出预测精确率（precision）、**召回率（recall，即准确率）**、F1 score 和各类别的样本量（support）。代码如下所示。

```
from sklearn.metrics import classification_report

y_pred_probabilities = model.predict(new_x_test)
y_pred = np.argmax(y_pred_probabilities, axis=-1)

# 使用 classification_report 函数计算每个类别的评估指标
report = classification_report(new_y_test, y_pred, digits=2,
output_dict=False)
print(report)
```

1.2.5 Argument Selection（参数选择）

为了对比不同数据集对模型性能的影响，我引入了 3 组不同的训练集，如表 1 所示。

表 1：训练集构成

组	标签为 0 的数据个数	标签为 1 的数据个数	是否随机打乱
1	50000	50000	是
2	50000	50000	否
3	10000	90000	是

产生这 3 组不同数据集的代码详见附录。

1.3 Experimental Results（实验结果）

为了保存每一轮 mini-batch 训练后的模型在训练集和测试集上的损失（loss），使用下面的语句运行 main.py，并将训练过程中程序输出到终端的结果保存在 output.log 文件中。

```
python main.py > output.log
```

据此，可以直接看出每次 mini-batch 训练后模型在训练集和测试集上的损失（Train loss 和 Test loss），以及其在训练集上的准确率 accuracy，如图 7 所示。由于日志文件很长，因此后续会将 loss 的变化趋势绘制为图片的形式进行展示。

图 7：output.log 记录了每个 mini-batch 训练后模型在训练集和测试集上的损失 loss

1.3.1 第一组实验结果

在 epoch=8、batch_size=8000 的条件下，训练完成后模型在训练集和测试集上的损失 loss 和准确率 acc 分别如图 8 所示。此时测试集的 accuracy 达到了 0.9778。

整个训练过程中，模型在训练集和测试集上的损失 loss 随 mini-batch 数量的变化如图 9 所示。

```
Train accuracy: 0.9860600233078003
Train loss: 0.244769298285245895
Test accuracy: 0.9777500295639038
Test loss: 0.2757080137729645
```

图 8：实验结果（第一组）

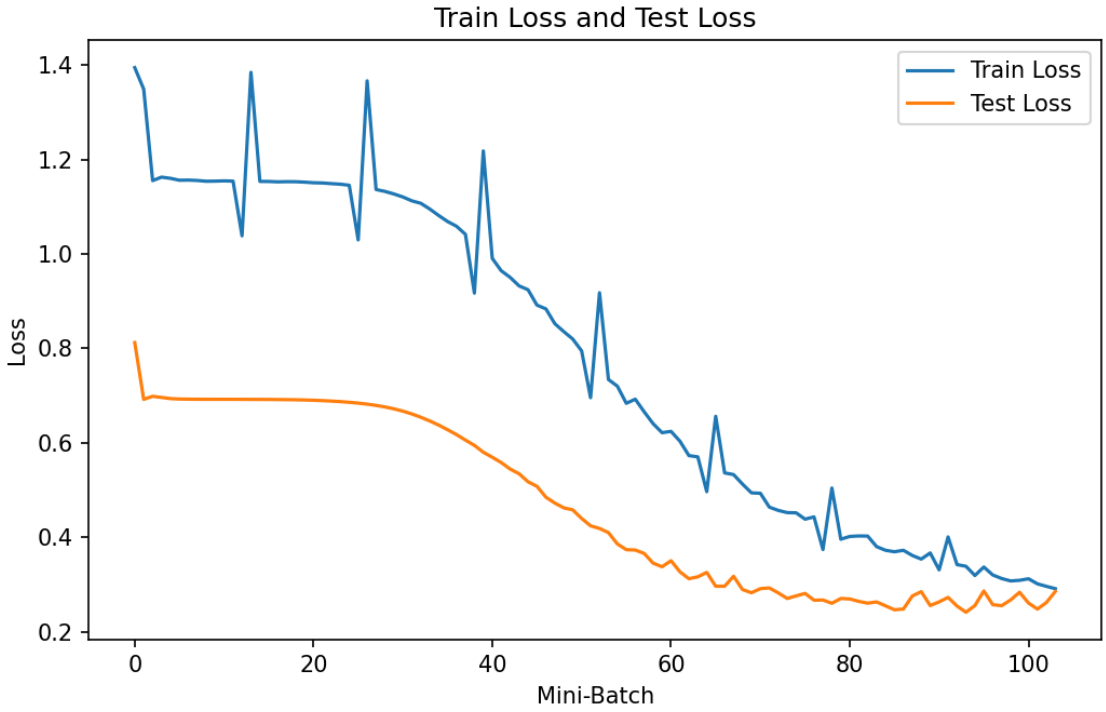

图 9：训练集和测试集上的损失 loss 随 mini-batch 数量的变化（第一组）

最终，模型在测试集二分类中每一类的准确率如图 10 所示。图中最左侧第一列为标签，第二列为精确率（precision），**第三列为召回率（recall，即所要求的准确率，代表模型识别出属于该类别的数据量与该类别在真实样本中的数量之比）**，后两列是 F1 score 和各标签对应的数据量（support）。

	precision	recall	f1-score	support
0	0.97	0.95	0.96	2000
1	0.99	0.98	0.98	2000
accuracy			0.97	4000
macro avg	0.98	0.97	0.97	4000
weighted avg	0.98	0.97	0.97	4000

图 10：测试集二分类中每一类的准确率（第一组）

1.3.2 第二组实验结果

保持第一组的模型架构和训练超参数不变，修改训练集的构造。此时，训练集标签为[0, 0, 0, 0, …, 1, 1, 1, 1]，即**前 50000 条数据的标签均为 0，后 50000 条数据的标签均为 1**。再次训练，训练完成后模型在训练集和测试集上的损失 loss 和准确率 acc 分别如图 11 所示。此时测试集的 accuracy 只有 0.4997。

整个训练过程中，模型在训练集和测试集上的损失 loss 随 mini-batch 数量的变化如图 12 所示。

```
Train accuracy: 0.979640007019043
Train loss: 0.03862805292010307
Test accuracy: 0.4997499883174896
Test loss: 2.6542415618896484
```

图 11：实验结果（第二组）

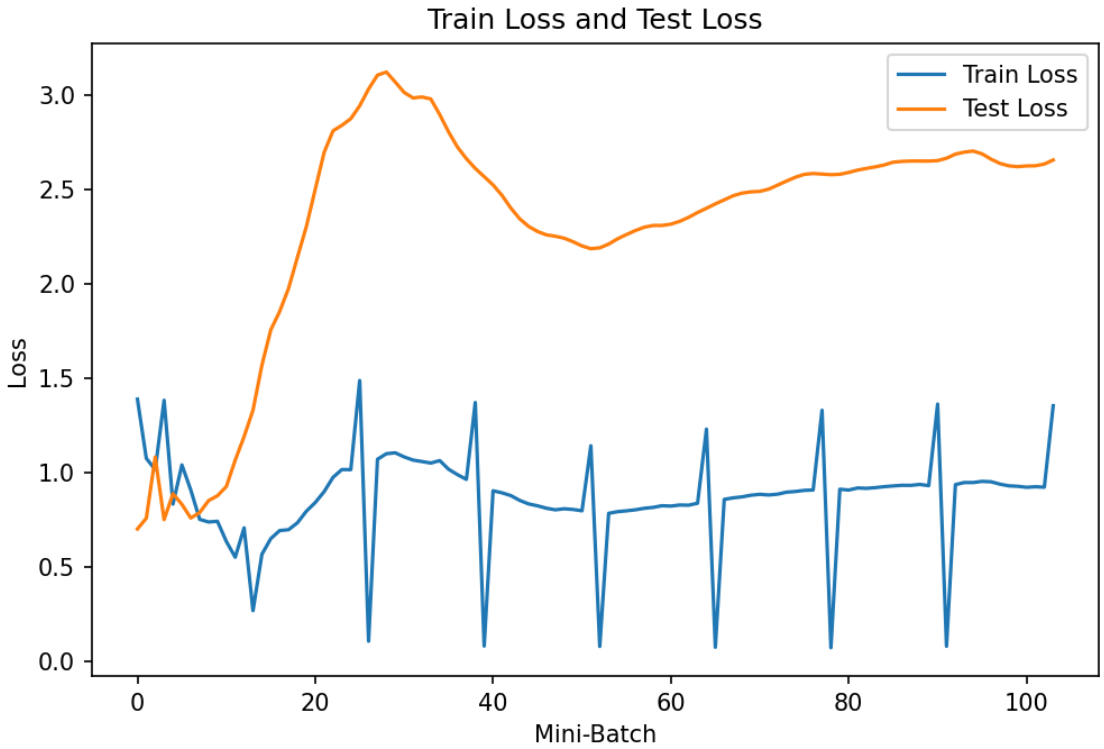

图 12：训练集和测试集上的损失 loss 随 mini-batch 数量的变化（第二组）

最终，模型在测试集二分类中每一类的准确率如图 13 所示。

	precision	recall	f1-score	support
0	0.50	0.35	0.41	2000
1	0.50	0.65	0.57	2000
accuracy			0.50	4000
macro avg	0.50	0.50	0.49	4000
weighted avg	0.50	0.50	0.49	4000

图 13：测试集二分类中每一类的准确率（第二组）

1.3.3 第三组实验结果

保持第一组的模型架构和训练超参数不变，修改训练集的构造。此时，**训练集中标签为 0 和标签为 1 的数据量分别为 10000 条和 90000 条**。再次训练，训练完成后模型在训练集和测试集上的损失 loss 和准确率 acc 分别如图 14 所示。此时测试集的 accuracy 只有 0.5450。

整个训练过程中，模型在训练集和测试集上的损失 loss 随 mini-batch 数量的变化如图 15 所示。

```
Train accuracy: 0.9787300229072571
Train loss: 0.052700214087963104
Test accuracy: 0.5450000166893005
Test loss: 3.9257946014404297
```

图 14：实验结果（第三组）

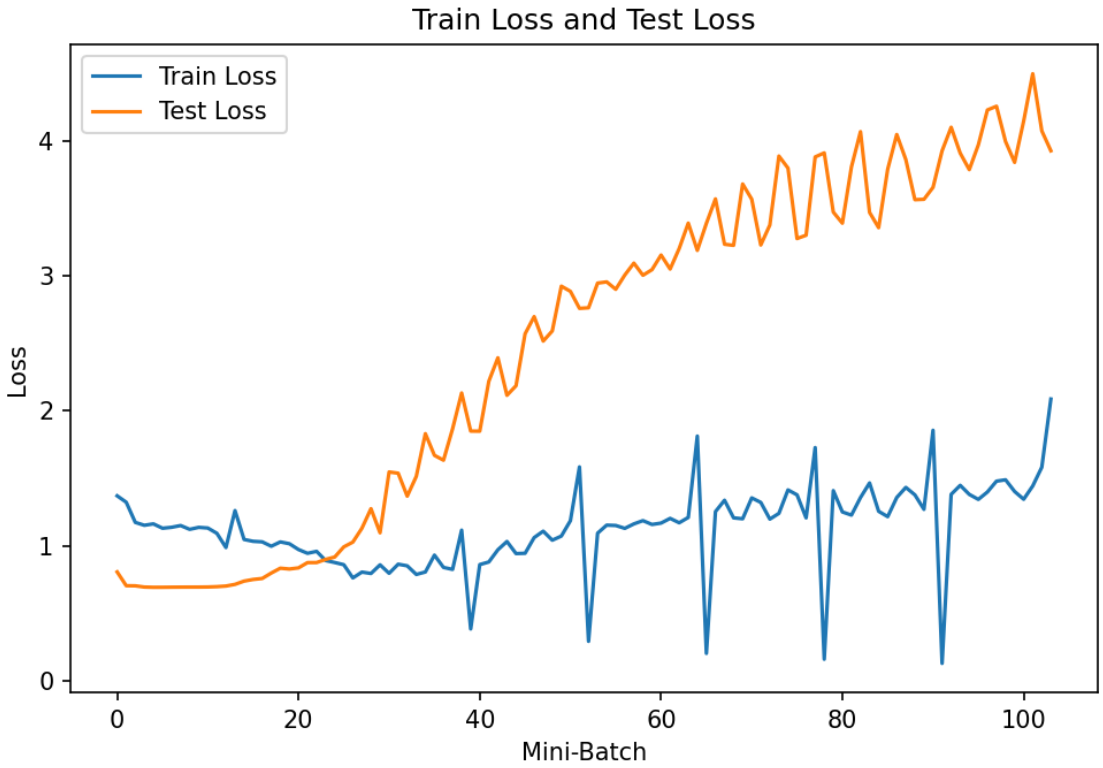

图 15：训练集和测试集上的损失 loss 随 mini-batch 数量的变化（第三组）

最终，模型在测试集二分类中每一类的准确率如图 16 所示。

	precision	recall	f1-score	support
0	0.97	0.09	0.17	2000
1	0.52	1.00	0.69	2000
accuracy			0.55	4000
macro avg	0.75	0.55	0.43	4000
weighted avg	0.75	0.55	0.43	4000

图 16：测试集二分类中每一类的准确率（第三组）

1.4 Discussion（结果分析）

1.4.1 总体结果分析

本实验中的模型（数据集和训练超参数合理的条件下，即第一组模型）**在测试集上整体的 accuracy 可以达到 0.97 以上，并且在标签为 0 和标签为 1 的两个类别中均达到了 0.97 的准确率**。这说明模型具有较高的稳定性和泛化性。

观察上面的训练集和测试集上的损失 loss 随 mini-batch 数量的变化趋势图像，可以发现对于图 9 而言，训练集上 loss 波动较大（蓝色线），而测试集上 loss 相对平滑（橙色线）。一种可能的解释是：ResNet 模型在训练时使用了残差连接，可以避免梯度消失问题，有助于训练非常深的神经网络。由于残差连接的存在，ResNet 模型在训练时拟合能力较强，导致在每两个 epoch 之间能更快地适应训练集的数据特征，在训练集上的损失发生突变。

本实验中，各卷积层的神经元个数均为 32，说明本实验中每个卷积层只需要 32 个神经元就可以提取绝大部分的有效特征了。

为了在测试集上获得较高的 accuracy，在数据选取上需要注意避免数据不均匀和数据不平衡的问题。第二组和第三组的模型就分别出现了这两种问题，下面会逐一进行分析。

1.4.2 数据不均匀分布导致的问题分析

第一组和第二组的主要区别在于第一组的训练集是随机打乱的，而第二组的训练集前 50000 个和后 50000 个分别为标签为 0 的数据和标签为 1 的数据，具有很强的规律性。

从结果上看，第一组在测试集上的准确率为 0.9778，而第二组的准确率仅有 0.4997，说明**第二组的模型产生了严重的过拟合现象**。如图 12 所示，**模型不仅在训练集上的 loss 降不下去，在测试集上的 loss 也迅速膨胀，出现了梯度消失问题**。其原因可以通过两个简单的例子说明：

- 假设训练集标签为[0, 1, 0, 1, 0, 1, …, 0, 1]，那么模型很可能会把“标签 0-1 交替出现”的特点作为特征学习到；
- 假设训练集标签为[0, 0, 0, 0, …, 1, 1, 1, 1]，由于训练时是以 mini-batch 为单位进行学习的，因此后期送入模型的数据的标签全为 1，模型很可能会把“标签全为 1”的特点作为特征学习到，最后预测出来的结果泛化性很差。

因此，根据上述分析，最合适的方式还是对训练集的数据进行随机打乱。

1.4.3 数据不平衡导致的问题分析

第一组和第三组的主要区别在于第一组的训练集中标签为 0 和标签为 1 的数据量均为 50000 条，而第三组的训练集中标签为 0 和标签为 1 的数据量分别为 10000 条和 90000 条，**两种类型的数据不平衡**。

从结果上看，第一组在测试集上的准确率为 0.9778，而第三组的准确率仅有 0.5450，说明第三组的模型的泛化性很差。由图 16 可知：

- **精确度**：表示在所有被分类为某个类别的样本中，被正确分类的比例。标签 0 的精确度为 0.97，标签 1 的精确度为 0.52。
- **召回率**：指在所有实际为某一类别的样本中，被正确预测为该类别的样本比例。标签 0 的召回率为 0.09，标签 1 的召回率为 1。

据此可以得出三条结论：

- **标签 0 的召回率非常低**，只有 9%。可见模型在识别真实为 0 的样本时存在较大缺陷，出现了大量的 False Negative。
- **标签 1 的精确度较低**，只有 52%。可见模型在将样本预测为 1 时会有相当一部分 False Positive，**但对于真实为 1 的样本有很好的识别能力**。
- 标签 1 的 F1-score 为 0.69，远大于标签 0 的 F1-score 0.17。因此**模型对于标签 1 的预测结果相对更准确**。

综上所述，**对于类别占比较多的类型数据，模型将有更多的样本用于学习该类别的特征和模式，从而在一定程度上提高其召回率；反之，对于类别占比较少的类型数据，可能导致模型在训练过程中更难以学习到该类别的特征，模型更倾向于将有限的预测机会用在真实为该类别的样本上，因此其精确率更高**。以后在面对类似本实验的问题中，应当时刻关注数据是否平衡，如果不平衡则需要手动调整训练集的数据构成，通过数据增强的方式确保数据的平衡性。

数据增强有多种方式实现：

- 在原始数据足够多的情况下，可以直接通过限制不同类别数据集的数量达到数据的平衡性。（本实验中属于这种情况）
- 在原始数据很少的情况下，可以通过旋转（rotation）、翻转（flip）、裁剪（crop）等方式实现数据增强。

1.5 Reference（参考文献）

[1] 华中科技大学计算机学院 2023 年秋季计算机视觉《第五讲 卷积神经网络（上）》
[2] 华中科技大学计算机学院 2023 年秋季计算机视觉《第五讲 卷积神经网络（下）》
[3] 华中科技大学计算机学院 2023 年秋季计算机视觉《第五讲 常见 CNN 网络及深度学习平台 v2.0》
[4] NN-SVG https://github.com/alexlenail/NN-SVG

1.6 Appendix（源代码）

1.6.1 main.py

作用：定义卷积神经网络和 ResNet 模块，尝试不同的数据集构成，训练模型。
代码：

```
import tensorflow as tf
import numpy as np
from tensorflow.keras.callbacks import Callback

class LossHistory(Callback):
    def __init__(self, train_data, test_data):
        self.train_data = train_data
        self.test_data = test_data

    def on_batch_end(self, batch, logs=None):
        train_loss = logs.get('loss')
        test_loss = self.model.evaluate(self.test_data[0],
self.test_data[1], verbose=0)
        print("    Mini-Batch:", batch, "- Train Loss:", train_loss, " -
Test Loss:", test_loss[0])

    def on_epoch_end(self, epoch, logs=None):
        pass

mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

train_samples = int(len(x_train) * 0.1)
```

```
test_samples = int(len(x_test) * 0.1)

train_indices = np.random.choice(len(x_train), train_samples,
replace=False)
test_indices = np.random.choice(len(x_test), test_samples,
replace=False)

x_train = x_train[train_indices] / 255.0
y_train = y_train[train_indices]
x_test = x_test[test_indices] / 255.0
y_test = y_test[test_indices]

# 处理训练集(1)
label0 = 0
label1 = 0
flag = 0 # 当前寻找的标签
new_x_train = []
new_y_train = []
for i in range(0, len(x_train)-1):
    for j in range(i, len(x_train)-1):
        if flag == 1 and y_train[i] == y_train[j] and label1 < 50000:
            new_x_train.append([x_train[i], x_train[j]])
            new_y_train.append(1)
            label1 += 1
            flag = 0
        if flag == 0 and y_train[i] != y_train[j] and label0 < 50000:
            new_x_train.append([x_train[i], x_train[j]])
            new_y_train.append(0)
            label0 += 1
            flag = 1
        if label1 == 50000 and label0 == 50000:
            break

new_x_train = np.array(new_x_train)
new_y_train = np.array(new_y_train)

np.random.seed(0)
random_index = np.random.permutation(len(new_x_train))
new_x_train = new_x_train[random_index]
new_y_train = new_y_train[random_index]

# # 处理训练集(2)
# label0 = 0
# label1 = 0
```

```
# new_x_train = []
# new_y_train = []
#
# for i in range(0, len(x_train)-1):
#     for j in range(i, len(x_train)-1):
#         if label0 < 50000 and i < 50000:
#             new_x_train.append([x_train[i], x_train[j]])
#             new_y_train.append(0)
#             label0 += 1
#         elif label1 < 50000 and j >= len(x_train)-50000:
#             new_x_train.append([x_train[i], x_train[j]])
#             new_y_train.append(1)
#             label1 += 1
#         if label0 == 50000 and label1 == 50000:
#             break
#
# new_x_train = np.array(new_x_train)
# new_y_train = np.array(new_y_train)
#
# np.random.seed(0)
# random_index = np.random.permutation(len(new_x_train))
# new_x_train = new_x_train[random_index]
# new_y_train = new_y_train[random_index]

# # 处理训练集(3)
# label0 = 0
# label1 = 0
# flag = 0 # 当前寻找的标签
# new_x_train = []
# new_y_train = []
# for i in range(0, len(x_train)-1):
#     for j in range(i, len(x_train)-1):
#         if flag == 1 and y_train[i] == y_train[j] and label1 < 90000:
#             new_x_train.append([x_train[i], x_train[j]])
#             new_y_train.append(1)
#             label1 += 1
#             flag = 0
#             if label0 == 10000:
#                 flag = 1
#         if flag == 0 and y_train[i] != y_train[j] and label0 < 10000:
#             new_x_train.append([x_train[i], x_train[j]])
#             new_y_train.append(0)
#             label0 += 1
#             flag = 1
```

```
#         if label1 == 90000 and label0 == 10000:
#             break
#
# new_x_train = np.array(new_x_train)
# new_y_train = np.array(new_y_train)
#
# np.random.seed(0)
# random_index = np.random.permutation(len(new_x_train))
# new_x_train = new_x_train[random_index]
# new_y_train = new_y_train[random_index]

# 处理测试集
label0 = 0
label1 = 0
flag = 0 # 当前寻找的标签
new_x_test = []
new_y_test = []
for i in range(0, len(x_test)-1):
    for j in range(i, len(x_test)-1):
        if flag == 1 and y_test[i] == y_test[j] and label1 < 2000:
            new_x_test.append([x_test[i], x_test[j]])
            new_y_test.append(1)
            label1 += 1
            flag = 0
        if flag == 0 and y_test[i] != y_test[j] and label0 < 2000:
            new_x_test.append([x_test[i], x_test[j]])
            new_y_test.append(0)
            label0 += 1
            flag = 1
        if label1 == 2000 and label0 == 2000:
            break

new_x_test = np.array(new_x_test)
new_y_test = np.array(new_y_test)

np.random.seed(0)
random_index = np.random.permutation(len(new_x_test))
new_x_test = new_x_test[random_index]
new_y_test = new_y_test[random_index]

count_0 = np.count_nonzero(new_y_train == 0)
count_1 = np.count_nonzero(new_y_train == 1)

print("新的训练集中标签为 0 的数量：", count_0)
```

```
print("新的训练集中标签为 1 的数量：", count_1)
print("新的训练集数据形状：", new_x_train.shape)
print("新的训练集标签形状：", new_y_train.shape)
print("新的测试集数据形状：", new_x_test.shape)
print("新的测试集标签形状：", new_y_test.shape)

# 调整输入形状
new_x_train = new_x_train.reshape(-1, 56, 28, 1)
new_x_test = new_x_test.reshape(-1, 56, 28, 1)

# 创建卷积神经网络模型
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(64, 3, activation='relu', input_shape=(56, 28,
1)),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Conv2D(128, 3, activation='relu'),
    tf.keras.layers.MaxPooling2D(),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(2, activation='sigmoid')
])

# 计算类别权重
total_samples = len(new_y_train)
class_weight = {0: total_samples / np.sum(new_y_train == 0), 1:
total_samples / np.sum(new_y_train == 1)}

model.compile(optimizer='adam',

loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
history = model.fit(new_x_train, new_y_train, epochs=8, batch_size=8000,
class_weight=class_weight, callbacks=[LossHistory((new_x_train,
new_y_train), (new_x_test, new_y_test))])

train_loss, train_acc = model.evaluate(new_x_train, new_y_train)
test_loss, test_acc = model.evaluate(new_x_test, new_y_test)
print('\nTrain accuracy:', train_acc)
print('\nTrain loss:', train_loss)
print('\nTest accuracy:', test_acc)
print('\nTest loss:', test_loss)
```

```
predictions = model.predict(new_x_test)
predictions = np.argmax(predictions, axis=1)
print("\nPredictions:", predictions)

import numpy as np
from sklearn.metrics import classification_report

y_pred_probabilities = model.predict(new_x_test)
y_pred = np.argmax(y_pred_probabilities, axis=-1)

report = classification_report(new_y_test, y_pred, digits=2,
output_dict=False)
print(report)
```

1.6.2 extract.py

作用：使用正则表达式提取每个 mini-batch 训练后在训练集和测试集上的损失。

代码：

```
import re
import csv
import chardet

input_file = "output.log"
output_file = "Loss.csv"
loss_values = []

with open(input_file, 'rb') as file:
    result = chardet.detect(file.read())
encoding = result['encoding']

with open(input_file, 'r', encoding=encoding) as file:
    for line in file:
        match = re.search(r'Train Loss: (\d+\.\d+)  - Test Loss:
(\d+\.\d+)', line)
        if match:
            train_loss = float(match.group(1))
            test_loss = float(match.group(2))
            loss_values.append([train_loss, test_loss])

with open(output_file, 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(['Train Loss', 'Test Loss'])
```

```
    for values in loss_values:
        writer.writerow(values)
```

1.6.3 draw.py

作用：绘制 Train Loss 和 Test Loss 的变化趋势图。
代码：

```
import csv
import matplotlib.pyplot as plt

input_file = 'Loss.csv'
train_loss = []
test_loss = []

with open(input_file, 'r') as file:
    reader = csv.reader(file)
    next(reader)
    for row in reader:
        train_loss.append(float(row[0]))
        test_loss.append(float(row[1]))

plt.plot(train_loss, label='Train Loss')
plt.plot(test_loss, label='Test Loss')
plt.xlabel('Mini-Batch')
plt.ylabel('Loss')
plt.title('Train Loss and Test Loss')
plt.legend()

plt.show()
```